BOOK 10: ERROR HANDLING
Chapter 1: Understanding Errors

WHAT ARE ERRORS

Errors are problems that stop your code from running:

console.log(hello)

Output: ReferenceError: hello is not defined

JavaScript could not find a variable called hello.

WHY ERRORS HAPPEN

Common causes:

Typos in variable or function names
Using undefined values
Calling methods on wrong types
Network failures
Invalid user input
Logic mistakes

Errors are normal. Every programmer creates them.

ERROR TYPES

JavaScript has built-in error types:

ReferenceError - using undefined variable:

```
console.log(notDefined)
```

TypeError - wrong type or method:

```
let num = 5
num.toUpperCase()

null.toString()
```

SyntaxError - invalid code structure:

```
let x =

if (true {
}
```

RangeError - value out of allowed range:

```
let arr = new Array(-1)

(1).toFixed(101)
```

THE ERROR OBJECT

Errors are objects with properties:

```
try {
 throw new Error("Something went wrong")
} catch (error) {
 console.log(error.name)
 console.log(error.message)
 console.log(error.stack)
}
```

name - type of error
message - description of the error
stack - where the error occurred

READING ERROR MESSAGES

Error message format:

```
TypeError: Cannot read properties of undefined (reading 'name')
 at getUser (app.js:15:20)
 at main (app.js:25:5)
 at app.js:30:1
```

First line: error type and description
Remaining lines: stack trace showing function calls

The stack trace shows:
- getUser was called at app.js line 15
- main called getUser at app.js line 25
- main was called at app.js line 30

Read from top to bottom. The first line is where the error occurred.

COMMON ERROR PATTERNS

Using undefined:

```
let user
console.log(user.name)
```

Accessing property of undefined. Check if variable exists first.

Wrong array index:

```
let items = ["a", "b", "c"]
console.log(items[5])
```

Output: undefined

No error, but you get undefined.

Calling non-function:

```
let result = "hello"()
```

Output: TypeError: "hello" is not a function

Null reference:

```
let element = document.getElementById("missing")
element.style.color = "red"
```

If element is null, you get TypeError.

SYNCHRONOUS VS ASYNCHRONOUS ERRORS

Synchronous errors happen immediately:

```
let x = JSON.parse("invalid json")
```

Asynchronous errors happen later:

```
setTimeout(function() {
 throw new Error("Async error")
}, 1000)
```

Async errors are harder to catch. You need special handling.

SILENT FAILURES

Some operations fail silently:

```
let obj = Object.freeze({ name: "Alice" })
obj.name = "Bob"

console.log(obj.name)
```

Output: Alice

The assignment silently failed because the object is frozen.

In strict mode, this throws an error:

```
"use strict"
let obj = Object.freeze({ name: "Alice" })
obj.name = "Bob"
```

STRICT MODE

"use strict" catches more errors:

```
"use strict"

x = 10
```

Using a variable without declaring it throws an error.

```
"use strict"

function example(a, a) {
}
```

Duplicate parameter names throw an error.

Add "use strict" at the top of your files.

DEBUGGING ERRORS

Console methods help debug:

console.log(variable)

console.error("Error message")

console.warn("Warning message")

console.table(arrayOrObject)

console.trace()

console.trace() shows the stack trace at any point.

BROWSER DEVELOPER TOOLS

Press F12 to open developer tools.

Console tab shows errors and logs.
Sources tab lets you set breakpoints.
Network tab shows API errors.

Click an error message to see where it occurred.

PRACTICE

Read and understand different error types.

Examine the stack trace of an error.

Use console methods for debugging.

Enable strict mode in your code.

Identify common error patterns.

WHAT YOU LEARNED

You learned errors stop code execution.
You learned JavaScript has different error types.
You learned to read error messages and stack traces.
You learned common error patterns.
You learned about strict mode.
You learned debugging tools.

The next chapter teaches you try/catch for handling errors.

BOOK 10: ERROR HANDLING
Chapter 2: Try/Catch

TRY AND CATCH

Wrap risky code in try block:

```
try {
 let result = JSON.parse("invalid json")
 console.log(result)
} catch (error) {
 console.log("Parse failed:", error.message)
}

console.log("Code continues")
```

Output:
Parse failed: Unexpected token i in JSON at position 0

Code continues

Without try/catch, the error would stop execution.

HOW TRY/CATCH WORKS

1. Code in try block runs
2. If no error, catch block is skipped
3. If error, execution jumps to catch
4. Code after try/catch continues

```
try {
 console.log("This runs")
 let x = undefined.property
 console.log("This does not run")
} catch (error) {
 console.log("Caught:", error.message)
}

console.log("This runs too")
```

THE CATCH PARAMETER

The catch block receives the error object:

```
try {
 throw new Error("Custom error")
} catch (error) {
 console.log(error.name)
 console.log(error.message)
 console.log(error.stack)
}
```

You can name the parameter anything:

```
try {
 riskyOperation()
} catch (e) {
 console.log(e.message)
}
```

Optional catch binding (ES2019):

```
try {
 riskyOperation()
} catch {
 console.log("Something went wrong")
}
```

THE FINALLY BLOCK

Always runs, with or without error:

```
try {
 console.log("Trying...")
 throw new Error("Oops")
} catch (error) {
 console.log("Caught:", error.message)
} finally {
 console.log("Finally runs")
}
```

Output:
Trying...
Caught: Oops
Finally runs

Finally runs even if catch throws:

```
try {
 throw new Error("First")
```

```
} catch (error) {
 throw new Error("Second")
} finally {
 console.log("Still runs")
}
```

USING FINALLY FOR CLEANUP

```
function processFile() {
 let file = openFile("data.txt")

 try {
 let data = readFile(file)
 return processData(data)
 } finally {
 closeFile(file)
 }
}
```

The file is always closed, even if an error occurs.

Common cleanup tasks:
- Closing connections
- Hiding loading spinners
- Releasing resources
- Resetting state

THROWING ERRORS

Create your own errors with throw:

```
function divide(a, b) {
 if (b === 0) {
 throw new Error("Cannot divide by zero")
```

```
 }
 return a / b
}

try {
 let result = divide(10, 0)
} catch (error) {
 console.log(error.message)
}
```

Output: Cannot divide by zero

THROWING DIFFERENT VALUES

You can throw anything:

```
throw "Error message"

throw 404

throw { code: "INVALID_INPUT", message: "Bad data" }

throw new Error("Standard error")
```

Use Error objects for consistency. They have stack traces.

RETHROWING ERRORS

Catch, log, and rethrow:

```
function processOrder(order) {
 try {
 validateOrder(order)
 chargeCustomer(order)
```

```
 shipOrder(order)
 } catch (error) {
 console.error("Order processing failed:", error)
 throw error
 }
}
```

The error is logged and then thrown to the caller.

CONDITIONAL CATCHING

Handle different error types differently:

```
try {
 riskyOperation()
} catch (error) {
 if (error instanceof TypeError) {
 console.log("Type error:", error.message)
 } else if (error instanceof RangeError) {
 console.log("Range error:", error.message)
 } else {
 throw error
 }
}
```

Rethrow errors you cannot handle.

NESTED TRY/CATCH

```
try {
 try {
 throw new Error("Inner error")
 } catch (innerError) {
 console.log("Inner caught:", innerError.message)
```

```
 throw new Error("Outer error")
 }
} catch (outerError) {
 console.log("Outer caught:", outerError.message)
}
```

Output:
Inner caught: Inner error
Outer caught: Outer error

TRY/CATCH WITH FUNCTIONS

Errors propagate up the call stack:

```
function a() {
 throw new Error("Error in a")
}

function b() {
 a()
}

function c() {
 b()
}

try {
 c()
} catch (error) {
 console.log("Caught:", error.message)
}
```

The error from a() propagates through b() and c() until caught.

PRACTICAL EXAMPLES

Safe JSON parsing:

```
function parseJSON(text, fallback = null) {
 try {
 return JSON.parse(text)
 } catch (error) {
 console.warn("Invalid JSON:", text)
 return fallback
 }
}

let data = parseJSON('{"valid": true}')
console.log(data)

let bad = parseJSON("not json", {})
console.log(bad)
```

Safe property access:

```
function getNestedValue(obj, path, defaultValue) {
 try {
 let value = path.split(".").reduce(function(current, key) {
 return current[key]
 }, obj)

 return value !== undefined ? value : defaultValue
 } catch (error) {
 return defaultValue
 }
}

let user = { profile: { name: "Alice" } }

console.log(getNestedValue(user, "profile.name", "Unknown"))
console.log(getNestedValue(user, "profile.age", 0))
console.log(getNestedValue(user, "settings.theme", "light"))
```

Form validation:

```
function validateForm(data) {
 let errors = []

 try {
 if (!data.email) {
 throw new Error("Email is required")
 }

 if (!data.email.includes("@")) {
 throw new Error("Invalid email format")
 }
 } catch (error) {
 errors.push({ field: "email", message: error.message })
 }

 try {
 if (!data.password) {
 throw new Error("Password is required")
 }

 if (data.password.length < 8) {
 throw new Error("Password too short")
 }
 } catch (error) {
 errors.push({ field: "password", message: error.message })
 }

 return errors
}
```

WHEN NOT TO USE TRY/CATCH

Do not wrap everything:

```
try {
 let x = 1 + 2
```

```
} catch (error) {
}
```

This code cannot fail. Try/catch is unnecessary.

Use try/catch for:
- Parsing data (JSON, dates)
- External operations (network, files)
- User input processing
- Third-party libraries

PRACTICE

Wrap risky code in try/catch.

Use finally for cleanup.

Throw custom errors.

Handle different error types.

Create safe wrapper functions.

WHAT YOU LEARNED

You learned try/catch prevents error crashes.
You learned catch receives the error object.
You learned finally always runs.
You learned throw creates errors.
You learned to rethrow errors you cannot handle.
You learned when to use try/catch.

The next chapter teaches you about custom error types.
BOOK 10: ERROR HANDLING
Chapter 3: Custom Errors

WHY CUSTOM ERRORS

Built-in errors are generic:

throw new Error("Something went wrong")

Custom errors are specific:

throw new ValidationError("Email is invalid", "email")
throw new NetworkError("Server unreachable", 503)
throw new AuthenticationError("Invalid token")

Custom errors help you handle different problems differently.

CREATING CUSTOM ERRORS

Extend the Error class:

```
class ValidationError extends Error {
 constructor(message) {
 super(message)
 this.name = "ValidationError"
 }
}
```

throw new ValidationError("Invalid email format")

The error has:
- Custom name "ValidationError"
- Your message
- Stack trace

ADDING PROPERTIES

```
class ValidationError extends Error {
 constructor(message, field) {
 super(message)
 this.name = "ValidationError"
 this.field = field
 }
}

let error = new ValidationError("Required field", "email")

console.log(error.name)
console.log(error.message)
console.log(error.field)
```

Output:
ValidationError
Required field
email

MULTIPLE CUSTOM ERRORS

```
class AppError extends Error {
 constructor(message) {
 super(message)
 this.name = this.constructor.name
 }
}

class ValidationError extends AppError {
 constructor(message, field) {
 super(message)
```

```
 this.field = field
 }
}

class NetworkError extends AppError {
 constructor(message, statusCode) {
 super(message)
 this.statusCode = statusCode
 }
}

class AuthError extends AppError {
 constructor(message) {
 super(message)
 }
}
```

All errors share a base class. Each has specific properties.

HANDLING CUSTOM ERRORS

```
try {
 processOrder(order)
} catch (error) {
 if (error instanceof ValidationError) {
 showFieldError(error.field, error.message)
 } else if (error instanceof NetworkError) {
 if (error.statusCode === 503) {
 showRetryButton()
 } else {
 showNetworkError(error.message)
 }
 } else if (error instanceof AuthError) {
 redirectToLogin()
 } else {
 showGenericError()
 console.error("Unexpected error:", error)
```

```
 }
}
```

Different error types get different handling.

ERROR CODES

Use codes for programmatic handling:

```
class AppError extends Error {
 constructor(message, code) {
 super(message)
 this.name = this.constructor.name
 this.code = code
 }
}

class UserError extends AppError {}

let errors = {
 USER_NOT_FOUND: new UserError("User not found",
"USER_NOT_FOUND"),
 USER_INACTIVE: new UserError("User is inactive",
"USER_INACTIVE"),
 USER_BANNED: new UserError("User is banned", "USER_BANNED")
}

function getUser(id) {
 let user = database.find(id)

 if (!user) {
 throw errors.USER_NOT_FOUND
 }

 if (!user.active) {
 throw errors.USER_INACTIVE
 }
```

```
 return user
}

try {
 let user = getUser(123)
} catch (error) {
 switch (error.code) {
 case "USER_NOT_FOUND":
 showNotFound()
 break
 case "USER_INACTIVE":
 showReactivation()
 break
 default:
 showError(error.message)
 }
}
```

HTTP ERRORS

```
class HttpError extends Error {
 constructor(message, statusCode) {
 super(message)
 this.name = "HttpError"
 this.statusCode = statusCode
 }
}

class NotFoundError extends HttpError {
 constructor(message = "Resource not found") {
 super(message, 404)
 }
}

class UnauthorizedError extends HttpError {
 constructor(message = "Unauthorized") {
```

```
 super(message, 401)
 }
}

class ForbiddenError extends HttpError {
 constructor(message = "Forbidden") {
 super(message, 403)
 }
}

class ServerError extends HttpError {
 constructor(message = "Internal server error") {
 super(message, 500)
 }
}

async function handleResponse(response) {
 if (!response.ok) {
 switch (response.status) {
 case 404:
 throw new NotFoundError()
 case 401:
 throw new UnauthorizedError()
 case 403:
 throw new ForbiddenError()
 default:
 throw new HttpError("Request failed", response.status)
 }
 }

 return response.json()
}
```

VALIDATION ERRORS

```
class ValidationError extends Error {
 constructor(errors) {
```

```
 super("Validation failed")
 this.name = "ValidationError"
 this.errors = errors
 }

 getFieldErrors(field) {
 return this.errors.filter(function(e) {
 return e.field === field
 })
 }

 hasErrors() {
 return this.errors.length > 0
 }
}

function validateUser(data) {
 let errors = []

 if (!data.email) {
 errors.push({ field: "email", message: "Email is required" })
 } else if (!data.email.includes("@")) {
 errors.push({ field: "email", message: "Invalid email format" })
 }

 if (!data.password) {
 errors.push({ field: "password", message: "Password is required" })
 } else if (data.password.length < 8) {
 errors.push({ field: "password", message: "Password too short" })
 }

 if (errors.length > 0) {
 throw new ValidationError(errors)
 }

 return true
}

try {
 validateUser({ email: "test", password: "123" })
```

```
} catch (error) {
 if (error instanceof ValidationError) {
 error.errors.forEach(function(e) {
 showFieldError(e.field, e.message)
 })
 }
}
```

ERROR FACTORY

```
function createError(type, message, details = {}) {
 let error

 switch (type) {
 case "validation":
 error = new ValidationError(message)
 break
 case "network":
 error = new NetworkError(message)
 break
 case "auth":
 error = new AuthError(message)
 break
 default:
 error = new AppError(message)
 }

 Object.assign(error, details)
 return error
}

throw createError("validation", "Invalid input", { field: "email" })
```

WRAPPING EXTERNAL ERRORS

Wrap library errors in your own:

```
async function fetchData(url) {
 try {
 let response = await fetch(url)

 if (!response.ok) {
 throw new NetworkError(
 `Request failed: ${response.statusText}`,
 response.status
 )
 }

 return await response.json()
 } catch (error) {
 if (error instanceof NetworkError) {
 throw error
 }

 throw new NetworkError(
 "Network request failed: " + error.message,
 0
 )
 }
}
```

Your code always throws NetworkError, regardless of what fetch throws.

SERIALIZING ERRORS

Convert errors to JSON for logging or sending:

```
class AppError extends Error {
 constructor(message, code) {
 super(message)
```

```
 this.name = this.constructor.name
 this.code = code
 this.timestamp = new Date().toISOString()
 }

 toJSON() {
 return {
 name: this.name,
 message: this.message,
 code: this.code,
 timestamp: this.timestamp,
 stack: this.stack
 }
 }
}

let error = new AppError("Something failed",
"UNKNOWN_ERROR")

console.log(JSON.stringify(error.toJSON(), null, 2))
```

PRACTICE

Create a custom error class.

Add properties to custom errors.

Create an error hierarchy.

Handle different error types with instanceof.

Create error codes for programmatic handling.

WHAT YOU LEARNED

You learned to extend Error for custom types.
You learned to add properties to errors.
You learned to create error hierarchies.
You learned to use instanceof for error handling.
You learned error codes and factories.
You learned to wrap external errors.

The next chapter teaches you debugging techniques.

BOOK 10: ERROR HANDLING

Chapter 4: Debugging

CONSOLE METHODS

console.log - general output:

```
console.log("Value:", x)
console.log("User:", user)
```

console.error - error output (red in console):

```
console.error("Failed to load data")
console.error(error)
```

console.warn - warning output (yellow in console):

```
console.warn("Deprecated function used")
```

console.info - informational output:

```
console.info("App started")
```

FORMATTED OUTPUT

console.table - display arrays and objects as tables:

```
let users = [
 { name: "Alice", age: 25 },
 { name: "Bob", age: 30 }
]

console.table(users)
```

console.dir - display object properties:

```
let element = document.getElementById("main")
console.dir(element)
```

console.group - group related logs:

```
console.group("User Data")
console.log("Name:", user.name)
console.log("Email:", user.email)
console.log("Age:", user.age)
console.groupEnd()
```

console.count - count how many times code runs:

```
function processItem(item) {
 console.count("processItem called")
}
```

TIMING

console.time and console.timeEnd:

```
console.time("fetchData")

await fetchData()
```

console.timeEnd("fetchData")

Output: fetchData: 234.5ms

STACK TRACES

console.trace - show where code is running from:

```
function a() {
 b()
}

function b() {
 c()
}

function c() {
 console.trace("Reached c")
}

a()
```

Shows the call stack leading to c().

ASSERTIONS

console.assert - log only if condition is false:

```
let age = 15

console.assert(age >= 18, "User is underage:", age)
```

Only logs if the assertion fails.

BROWSER DEVELOPER TOOLS

Open with F12 or right-click and Inspect.

Console tab:
- View errors and logs
- Run JavaScript commands
- Filter by log level

Sources tab:
- View source code
- Set breakpoints
- Step through code

Network tab:
- View API requests
- Check response data
- See failed requests

BREAKPOINTS

Set breakpoints in Sources tab:

1. Click line number to add breakpoint
2. Code pauses when it reaches that line
3. Inspect variables in Scope panel
4. Step through code with controls

Stepping controls:
- Continue (F8) - run to next breakpoint
- Step over (F10) - execute line, skip function internals
- Step into (F11) - go into function call
- Step out (Shift + F11) - finish current function

CONDITIONAL BREAKPOINTS

Right-click line number and set condition:

x > 10

Breakpoint only triggers when condition is true.

DEBUGGER STATEMENT

Add breakpoint in code:

```
function processData(data) {
 debugger
 let result = transform(data)
 return result
}
```

When developer tools are open, code pauses at debugger.

INSPECTING VARIABLES

In paused state:

Scope panel shows local and global variables.
Watch panel lets you track specific expressions.
Console lets you evaluate expressions.

Hover over variables in source code to see values.

COMMON DEBUGGING PATTERNS

Binary search debugging:

```
function processData(data) {
 console.log("1: data received", data)

 let transformed = transform(data)
 console.log("2: transformed", transformed)

 let filtered = filter(transformed)
 console.log("3: filtered", filtered)

 let result = finalize(filtered)
 console.log("4: result", result)

 return result
}
```

Add logs to find where things go wrong. Remove half to narrow down.

Checking undefined:

```
function getUser(id) {
 console.log("getUser called with:", id, typeof id)

 let user = users.find(u => u.id === id)
 console.log("Found user:", user)

 return user
}
```

Log values and types to understand what is happening.

DEBUGGING ASYNC CODE

Async call stack:

```
async function loadPage() {
 try {
 let user = await fetchUser()
 console.log("User loaded:", user)

 let posts = await fetchPosts(user.id)
 console.log("Posts loaded:", posts)

 render(user, posts)
 } catch (error) {
 console.error("Error in loadPage:", error)
 console.error("Stack:", error.stack)
 }
}
```

Chrome shows async call stacks in developer tools.

DEBUGGING EVENTS

Log event details:

```
element.addEventListener("click", function(event) {
 console.log("Event type:", event.type)
 console.log("Target:", event.target)
 console.log("Current target:", event.currentTarget)
 console.log("Event:", event)
})
```

Monitor all events:

```
monitorEvents(element)

monitorEvents(element, "click")
```

unmonitorEvents(element)

Only works in Chrome console.

DEBUGGING DOM

Inspect elements:

```
let element = document.getElementById("main")
console.log(element)
console.dir(element)
```

Check computed styles:

```
let styles = getComputedStyle(element)
console.log("Width:", styles.width)
console.log("Display:", styles.display)
```

Watch DOM changes:

```
let observer = new MutationObserver(function(mutations) {
 mutations.forEach(function(mutation) {
 console.log("DOM changed:", mutation)
 })
})

observer.observe(element, {
 attributes: true,
 childList: true,
 subtree: true
})
```

DEBUGGING NETWORK REQUESTS

Log fetch requests:

```
async function fetchData(url) {
 console.log("Fetching:", url)

 let response = await fetch(url)
 console.log("Response status:", response.status)

 let data = await response.json()
 console.log("Response data:", data)

 return data
}
```

Use Network tab to see:
- Request URL and method
- Request headers and body
- Response status and data
- Timing information

COMMON MISTAKES

Variable shadowing:

```
let name = "Alice"

function greet() {
 let name = "Bob"
 console.log(name)
}

greet()
console.log(name)
```

Off-by-one errors:

```
let arr = [1, 2, 3]

for (let i = 0; i <= arr.length; i++) {
 console.log(arr[i])
}
```

Async timing issues:

```
let result

fetch("/api/data")
 .then(r => r.json())
 .then(data => result = data)

console.log(result)
```

PRACTICE

Use console methods to log values.

Set breakpoints and step through code.

Use debugger statement.

Debug an async function.

Find and fix a bug using debugging techniques.

WHAT YOU LEARNED

You learned console methods for logging.
You learned to use browser developer tools.
You learned to set breakpoints.
You learned to step through code.

You learned debugging patterns.
You learned to debug async code and events.

The next chapter teaches you defensive programming.

BOOK 10: ERROR HANDLING

Chapter 5: Defensive Programming

WHAT IS DEFENSIVE PROGRAMMING

Writing code that anticipates problems:

- Check inputs before using them
- Provide defaults for missing values
- Handle edge cases
- Fail gracefully

INPUT VALIDATION

Always validate function inputs:

```
function calculateDiscount(price, discountPercent) {
 if (typeof price !== "number" || price < 0) {
 throw new Error("Price must be a positive number")
 }

 if (typeof discountPercent !== "number" || discountPercent < 0 ||
discountPercent > 100) {
 throw new Error("Discount must be between 0 and 100")
 }

 return price * (discountPercent / 100)
}
```

DEFAULT VALUES

Use defaults for optional parameters:

```
function greet(name = "Guest") {
 console.log("Hello, " + name)
}

greet()
greet("Alice")
```

Default with object destructuring:

```
function createUser({ name = "Anonymous", age = 0, email = "" }
= {}) {
 return { name, age, email }
}

createUser()
createUser({ name: "Alice" })
createUser({ name: "Bob", age: 25 })
```

NULLISH COALESCING

Use ?? for null and undefined:

```
let name = user.name ?? "Unknown"

let count = settings.count ?? 10

let value = null ?? "default"

let zero = 0 ?? "default"
```

?? only uses default for null and undefined.
|| uses default for any falsy value.

OPTIONAL CHAINING

Safe property access with ?.:

```
let userName = user?.profile?.name

let length = array?.length

let result = object?.method?.()
```

Returns undefined instead of throwing if path does not exist.

Combine with nullish coalescing:

```
let name = user?.profile?.name ?? "Unknown"
```

GUARD CLAUSES

Return early for invalid cases:

```
function processOrder(order) {
 if (!order) {
 console.log("No order provided")
 return null
 }

 if (!order.items || order.items.length === 0) {
 console.log("Order has no items")
 return null
 }
```

```
 if (order.status !== "pending") {
 console.log("Order already processed")
 return null
 }

 return { processed: true }
}
```

Guard clauses make the main logic clearer.

TYPE CHECKING

```
function processData(data) {
 if (data === null || data === undefined) {
 throw new Error("Data is required")
 }

 if (typeof data !== "object") {
 throw new Error("Data must be an object")
 }

 if (!Array.isArray(data.items)) {
 throw new Error("Data.items must be an array")
 }

 return transform(data)
}
```

Check types before using values.

SAFE ARRAY OPERATIONS

Check array before accessing:

```
function getFirst(arr) {
 if (!Array.isArray(arr) || arr.length === 0) {
 return null
 }
 return arr[0]
}

function getLast(arr) {
 if (!Array.isArray(arr) || arr.length === 0) {
 return null
 }
 return arr[arr.length - 1]
}
```

Safe array methods:

```
let items = data?.items ?? []
let filtered = items.filter(i => i.active)
```

SAFE OBJECT ACCESS

```
function getNestedValue(obj, path, defaultValue = undefined) {
 if (!obj || typeof obj !== "object") {
 return defaultValue
 }

 let keys = path.split(".")
 let current = obj

 for (let key of keys) {
 if (current === null || current === undefined) {
 return defaultValue
 }
 current = current[key]
 }

 return current ?? defaultValue
```

```
}

let email = getNestedValue(user, "contact.email", "no email")
```

SAFE JSON PARSING

```
function safeJSONParse(text, defaultValue = null) {
 try {
 return JSON.parse(text)
 } catch (error) {
 return defaultValue
 }
}

let config = safeJSONParse(configString, {})
let data = safeJSONParse(apiResponse, [])
```

SAFE DOM ACCESS

```
function getElement(selector) {
 let element = document.querySelector(selector)

 if (!element) {
 console.warn("Element not found:", selector)
 }

 return element
}

function safeSetText(selector, text) {
 let element = document.querySelector(selector)

 if (element) {
 element.textContent = text
```

```
 }
}

safeSetText("#message", "Hello")
```

IMMUTABLE OPERATIONS

Avoid mutating inputs:

```
function addItem(array, item) {
 return [...array, item]
}

function removeItem(array, index) {
 return [...array.slice(0, index), ...array.slice(index + 1)]
}

function updateObject(obj, updates) {
 return { ...obj, ...updates }
}
```

Return new values instead of changing originals.

FREEZING OBJECTS

Prevent accidental modifications:

```
let config = Object.freeze({
 apiUrl: "https://api.example.com",
 timeout: 5000
})

config.apiUrl = "changed"
```

```
console.log(config.apiUrl)
```

Deep freeze:

```
function deepFreeze(obj) {
 Object.keys(obj).forEach(function(key) {
 if (typeof obj[key] === "object" && obj[key] !== null) {
 deepFreeze(obj[key])
 }
 })
 return Object.freeze(obj)
}
```

ASSERTIONS FOR DEVELOPMENT

```
function assert(condition, message) {
 if (!condition) {
 throw new Error("Assertion failed: " + message)
 }
}

function processItems(items) {
 assert(Array.isArray(items), "items must be an array")
 assert(items.length > 0, "items cannot be empty")

 return items.map(transform)
}
```

Remove assertions in production or use a build tool.

GRACEFUL DEGRADATION

```
function loadUserPreferences() {
 try {
```

```
 let saved = localStorage.getItem("preferences")
 return saved ? JSON.parse(saved) : getDefaultPreferences()
 } catch (error) {
 console.warn("Could not load preferences:", error)
 return getDefaultPreferences()
 }
}

function getDefaultPreferences() {
 return {
 theme: "light",
 language: "en",
 notifications: true
 }
}
```

When something fails, fall back to working behavior.

FAIL FAST

Catch problems early:

```
class User {
 constructor(name, email) {
 if (!name || typeof name !== "string") {
 throw new Error("Valid name required")
 }

 if (!email || !email.includes("@")) {
 throw new Error("Valid email required")
 }

 this.name = name.trim()
 this.email = email.trim().toLowerCase()
 }
}
```

Invalid data is rejected immediately.

PRACTICE

Add input validation to a function.

Use optional chaining and nullish coalescing.

Write guard clauses.

Create safe accessor functions.

Implement graceful degradation.

WHAT YOU LEARNED

You learned to validate inputs.
You learned to use default values.
You learned optional chaining and nullish coalescing.
You learned guard clauses for early returns.
You learned safe access patterns.
You learned graceful degradation.

The next chapter brings everything together with practice programs.
BOOK 10: ERROR HANDLING
Chapter 6: Practice Error Programs

ROBUST FORM VALIDATOR

Complete validation with custom errors:

```
class ValidationError extends Error {
 constructor(field, message, value) {
 super(message)
 this.name = "ValidationError"
 this.field = field
 this.value = value
 }
}

class FormValidator {
 constructor() {
 this.rules = {}
 this.errors = []
 }

 addRule(field, name, validator, message) {
 if (!this.rules[field]) {
 this.rules[field] = []
 }

 this.rules[field].push({ name, validator, message })
 }

 validate(data) {
 this.errors = []

 for (let field in this.rules) {
 let value = data[field]
 let fieldRules = this.rules[field]

 for (let rule of fieldRules) {
 try {
 let isValid = rule.validator(value, data)

 if (!isValid) {
 this.errors.push(new ValidationError(
 field,
 rule.message,
```

```
 value
 ))
 }
 } catch (error) {
 this.errors.push(new ValidationError(
 field,
 "Validation error: " + error.message,
 value
 ))
 }
 }
 }

 return this.errors.length === 0
 }

 getErrors() {
 return this.errors
 }

 getFieldErrors(field) {
 return this.errors.filter(e => e.field === field)
 }
}

let validator = new FormValidator()

validator.addRule("email", "required",
 value => value && value.trim() !== "",
 "Email is required"
)

validator.addRule("email", "format",
 value => /^[^\s@]+@[^\s@]+\.[^\s@]+$/.test(value),
 "Invalid email format"
)

validator.addRule("password", "required",
 value => value && value.length > 0,
 "Password is required"
```

```
)

validator.addRule("password", "minLength",
 value => value && value.length >= 8,
 "Password must be at least 8 characters"
)

validator.addRule("password", "strength",
 value => /[A-Z]/.test(value) && /[a-z]/.test(value) && /
\d/.test(value),
 "Password must contain uppercase, lowercase, and number"
)

validator.addRule("confirmPassword", "match",
 (value, data) => value === data.password,
 "Passwords do not match"
)

let formData = {
 email: "test@example.com",
 password: "weak",
 confirmPassword: "weak"
}

if (validator.validate(formData)) {
 console.log("Form is valid")
} else {
 validator.getErrors().forEach(error => {
 console.log(`${error.field}: ${error.message}`)
 })
}
```

RESILIENT API CLIENT

API client with retry, timeout, and error handling:

```
class ApiError extends Error {
```

```
 constructor(message, statusCode, data = null) {
 super(message)
 this.name = "ApiError"
 this.statusCode = statusCode
 this.data = data
 }
}

class NetworkError extends Error {
 constructor(message, originalError = null) {
 super(message)
 this.name = "NetworkError"
 this.originalError = originalError
 }
}

class TimeoutError extends Error {
 constructor(message, timeout) {
 super(message)
 this.name = "TimeoutError"
 this.timeout = timeout
 }
}

class ApiClient {
 constructor(baseUrl, options = {}) {
 this.baseUrl = baseUrl
 this.timeout = options.timeout ?? 10000
 this.retries = options.retries ?? 3
 this.retryDelay = options.retryDelay ?? 1000
 }

 async request(endpoint, options = {}) {
 let url = this.baseUrl + endpoint
 let lastError

 for (let attempt = 1; attempt <= this.retries; attempt++) {
 try {
 return await this.makeRequest(url, options)
 } catch (error) {
```

```
lastError = error
console.warn(`Attempt ${attempt} failed:`, error.message)

if (!this.shouldRetry(error)) {
throw error
}

if (attempt < this.retries) {
await this.delay(this.retryDelay * attempt)
}
}
}

throw lastError
}

async makeRequest(url, options) {
let controller = new AbortController()

let timeoutId = setTimeout(() => {
controller.abort()
}, this.timeout)

try {
let response = await fetch(url, {
...options,
signal: controller.signal,
headers: {
"Content-Type": "application/json",
...options.headers
}
})

clearTimeout(timeoutId)

if (!response.ok) {
let data = null
try {
data = await response.json()
} catch (e) {}
```

```
throw new ApiError(
data?.message ?? response.statusText,
response.status,
data
)
}

return await response.json()
} catch (error) {
clearTimeout(timeoutId)

if (error.name === "AbortError") {
throw new TimeoutError(
`Request timed out after ${this.timeout}ms`,
this.timeout
)
}

if (error instanceof ApiError) {
throw error
}

throw new NetworkError(
"Network request failed",
error
)
}
}

shouldRetry(error) {
if (error instanceof TimeoutError) return true
if (error instanceof NetworkError) return true
if (error instanceof ApiError) {
return error.statusCode >= 500
}
return false
}

delay(ms) {
```

```
 return new Promise(resolve = > setTimeout(resolve, ms))
 }

 async get(endpoint) {
 return this.request(endpoint)
 }

 async post(endpoint, data) {
 return this.request(endpoint, {
 method: "POST",
 body: JSON.stringify(data)
 })
 }
}

let api = new ApiClient("https://api.example.com", {
 timeout: 5000,
 retries: 3
})

try {
 let users = await api.get("/users")
 console.log("Users:", users)
} catch (error) {
 if (error instanceof TimeoutError) {
 console.log("Request timed out. Please try again.")
 } else if (error instanceof NetworkError) {
 console.log("Network error. Check your connection.")
 } else if (error instanceof ApiError) {
 console.log(`API error ${error.statusCode}: ${error.message}`)
 } else {
 console.log("Unknown error:", error.message)
 }
}
```

SAFE DATA TRANSFORMER

Transform data with comprehensive error handling:

```
class TransformError extends Error {
 constructor(message, field, originalValue) {
 super(message)
 this.name = "TransformError"
 this.field = field
 this.originalValue = originalValue
 }
}

class DataTransformer {
 constructor() {
 this.transforms = []
 this.errors = []
 }

 addTransform(field, transformer, options = {}) {
 this.transforms.push({
 field,
 transformer,
 required: options.required ?? false,
 defaultValue: options.defaultValue,
 errorMessage: options.errorMessage
 })
 }

 transform(data) {
 this.errors = []
 let result = {}

 for (let config of this.transforms) {
 let { field, transformer, required, defaultValue, errorMessage } =
config
 let value = this.getNestedValue(data, field)

 if (value === undefined || value === null) {
 if (required) {
 this.errors.push(new TransformError(
 errorMessage ?? `Missing required field: ${field}`,
```

```
field,
value
))
} else if (defaultValue !== undefined) {
this.setNestedValue(result, field, defaultValue)
}
continue
}

try {
let transformed = transformer(value, data)
this.setNestedValue(result, field, transformed)
} catch (error) {
this.errors.push(new TransformError(
errorMessage ?? `Failed to transform ${field}: ${error.message}`,
field,
value
))

if (defaultValue !== undefined) {
this.setNestedValue(result, field, defaultValue)
}
}
}

return result
}

getNestedValue(obj, path) {
return path.split(".").reduce((current, key) => {
return current?.[key]
}, obj)
}

setNestedValue(obj, path, value) {
let keys = path.split(".")
let current = obj

for (let i = 0; i < keys.length - 1; i++) {
if (!current[keys[i]]) {
```

```
 current[keys[i]] = {}
 }
 current = current[keys[i]]
 }

 current[keys[keys.length - 1]] = value
 }

 hasErrors() {
 return this.errors.length > 0
 }

 getErrors() {
 return this.errors
 }
}

let transformer = new DataTransformer()

transformer.addTransform("name", value => value.trim(), {
 required: true,
 errorMessage: "Name is required"
})

transformer.addTransform("email", value =>
value.toLowerCase().trim(), {
 required: true
})

transformer.addTransform("age", value => {
 let num = parseInt(value)
 if (isNaN(num)) throw new Error("Invalid number")
 return num
}, {
 defaultValue: 0
})

transformer.addTransform("settings.theme", value => value, {
 defaultValue: "light"
})
```

```
let rawData = {
 name: "  Alice  ",
 email: "ALICE@EXAMPLE.COM",
 age: "25"
}

let result = transformer.transform(rawData)

if (transformer.hasErrors()) {
 console.log("Transform errors:", transformer.getErrors())
}

console.log("Transformed:", result)
```

ERROR LOGGING SERVICE

Centralized error logging:

```
class ErrorLogger {
 constructor(options = {}) {
 this.endpoint = options.endpoint ?? "/api/errors"
 this.queue = []
 this.maxQueueSize = options.maxQueueSize ?? 100
 this.flushInterval = options.flushInterval ?? 5000
 this.context = {}

 this.startAutoFlush()
 this.setupGlobalHandlers()
 }

 setContext(context) {
 this.context = { ...this.context, ...context }
 }

 log(error, additionalContext = {}) {
 let entry = this.createEntry(error, additionalContext)
```

```
this.queue.push(entry)

if (this.queue.length >= this.maxQueueSize) {
this.flush()
}

console.error("[ErrorLogger]", entry)
}

createEntry(error, additionalContext) {
return {
timestamp: new Date().toISOString(),
name: error.name ?? "Error",
message: error.message ?? String(error),
stack: error.stack,
context: {
...this.context,
...additionalContext,
url: window.location?.href,
userAgent: navigator?.userAgent
}
}
}

async flush() {
if (this.queue.length === 0) return

let entries = [...this.queue]
this.queue = []

try {
await fetch(this.endpoint, {
method: "POST",
headers: { "Content-Type": "application/json" },
body: JSON.stringify({ errors: entries })
})
} catch (error) {
this.queue = [...entries, ...this.queue]
console.warn("Failed to send error logs:", error)
```

```
 }
 }

 startAutoFlush() {
 setInterval(() => this.flush(), this.flushInterval)
 }

 setupGlobalHandlers() {
 window.addEventListener("error", event => {
 this.log(event.error ?? new Error(event.message), {
 type: "uncaughtError",
 filename: event.filename,
 lineno: event.lineno,
 colno: event.colno
 })
 })

 window.addEventListener("unhandledrejection", event => {
 this.log(event.reason ?? new Error("Unhandled rejection"), {
 type: "unhandledRejection"
 })
 })
 }
}

let logger = new ErrorLogger({
 endpoint: "/api/errors",
 flushInterval: 10000
})

logger.setContext({
 appVersion: "1.0.0",
 userId: getCurrentUserId()
})

try {
 riskyOperation()
} catch (error) {
 logger.log(error, { operation: "riskyOperation" })
}
```

WHAT YOU BUILT

You built four complete programs:

Form validator with custom validation errors
Resilient API client with retry and timeout
Safe data transformer with defaults
Error logging service with global handlers

Each program demonstrated:

Custom error classes
Try/catch patterns
Defensive programming
Error aggregation and reporting
Graceful failure handling

WHAT COMES NEXT

You now understand error handling:

- Error types and messages (Book 10, Chapter 1)
- Try/catch/finally (Book 10, Chapter 2)
- Custom error classes (Book 10, Chapter 3)
- Debugging techniques (Book 10, Chapter 4)
- Defensive programming (Book 10, Chapter 5)

Book 11 teaches you modern JavaScript features. You will:

- Use ES6+ syntax and features
- Work with modules
- Understand destructuring and spread
- Use modern array methods

- Write cleaner code

Modern JavaScript makes development faster and cleaner.

You finished Book 10. You can handle errors gracefully. Now we learn modern JavaScript.